CHAPTER 8
Unsupervised Machine Learning

Introduction

Unsupervised learning is a key area within statistical machine learning that focuses on uncovering patterns and structures in unlabelled data. This includes techniques like clustering, dimensionality reduction, and generative modelling. Given that most real-world data is unstructured, extensive preprocessing is often required to transform it into a usable format, as discussed in previous chapters. The abundance of unstructured and unlabelled data makes unsupervised learning increasingly valuable. Unlike supervised learning, which relies on labelled examples and predefined target variables, unsupervised learning operates without such guidance. It can group similar items together, much like sorting a collection of coloured marbles into distinct clusters, or reduce complex datasets into simpler forms through dimensionality reduction, all without sacrificing important information. Evaluating the performance and generalization in unsupervised learning also requires different metrics compared to supervised learning.

Structure

In this chapter, we will discuss the following topics:

- Unsupervised learning
- Model selection and evaluation

Objectives

The objective of this chapter is to introduce unsupervised machine learning, ways to evaluate a trained unsupervised model. With real-world examples and tutorials to better explain and demonstrate the implementation.

Unsupervised learning

Unsupervised learning is a machine learning technique where algorithms are trained on unlabeled data without human guidance. The data has no predefined categories or labels and the goal is to discover patterns and hidden structures. Unsupervised learning works by finding similarities or differences in the data and grouping them into clusters or categories. For example, an unsupervised algorithm can analyze a collection of images and sort them by color, shape or size. This is useful when there is a lot of data and labeling them is difficult. For example, imagine you have a bag of 20 candies with various colors and shapes. You wish to categorize them into different groups, but you are unsure of the number of groups or their appearance. Unsupervised learning can help find the optimal way to sort or group items.

Another example is, let us take the iris dataset without flower type labels. Suppose from iris dataset you take a data of 100 flowers with different features, such as petal length, petal width, sepal length and sepal width. You want to group the flowers into different types, but you do not know how many types there are or what they look like. You

can use unsupervised learning to find the optimal number of clusters and assign each flower to one of them. You can use any of unsupervised learning algorithm, for example K-means algorithm for clustering, which is described in the **K-means** section. The algorithm will randomly be choosing K points as the centers of the clusters, and then assigning each flower to the nearest center. Then, it will update the centers by taking the average of the features of the flowers in each cluster. It will repeat this process until the clusters are stable and no more changes occur.

There are many unsupervised learning algorithms some most common ones are described in this chapter. Unsupervised learning models are used for three main tasks: clustering, association, and dimensionality reduction. *Table 8.1* summarizes these tasks:

Algorithm	Task	Description
K-means	Clustering	Divides data into a predefined number of clusters based on similarity.
K-prototype	Clustering	Similar to K-means, but can handle numerical, categorical, and text data.
Hierarchical clustering	Clustering	Creates a hierarchy of clusters by repeatedly merging or splitting groups of data points.
Gaussian mixture models	Clustering	Models data as a mixture of Gaussian distributions, allowing for more flexible clustering.
Principal component analysis	Dimensionality reduction	Finds a lower-dimensional representation of data while preserving as much information as possible.
Singular value decomposition	Dimensionality reduction	Factorizes a data matrix into three matrices, allowing for dimensionality reduction and data visualization.
DBSCAN	Clustering	Finds clusters of overlapping data points based on density.
t-Distributed	Dimensionality	Creates a two- or three-dimensional

Stochastic Neighbor Embedding (t-SNE)	reduction	representation of high-dimensional data while preserving local relationships.
Autoencoders	Dimensionality reduction and dimensionality increase	Learn a compressed representation of data and then reconstruct the original data, allowing for dimensionality reduction or dimensionality increase.
Apriori	Association	Uncovers frequent item sets in transactional datasets.
Eclat	Association	Similar to Apriori, but uses a more efficient algorithm for large datasets.
FP-Growth	Association	A more memory-efficient algorithm for finding frequent item sets.

***Table 8.1**: Summary of unsupervised learning algorithms and their tasks*

As described in *Table 8.1*, the primary applications of unsupervised learning include clustering, dimensionality reduction, and association rule mining. Association rule mining aims to uncover interesting relationships between items in a dataset, similar to identifying patterns in grocery shopping lists. High-dimensional data can be overwhelming, but dimensionality reduction simplifies it while retaining the most important information.

K-means

K-means clustering is an iterative algorithm that divides data points into a predefined number of clusters. It works by first randomly selecting K centroids, one for each cluster. It then assigns each data point to the nearest centroid. The centroids are then updated to be the average of the data points in their respective clusters. This process is repeated until the centroids no longer change. It is used to cluster numerical data. It is often used in marketing to segment customers, in finance to detect fraud and in data mining to discover hidden patterns in data.

For example, K-means can be applied here. Imagine you have a shopping cart dataset of items purchased by customers. You want to group customers into clusters based on the items they tend to buy together.

Before moving to the tutorials let us look at the syntax for implementing K-means with sklearn, which is as follows*:*

```
from sklearn.cluster import KMeans
# Load the dataset
data = ...
# Create and fit the k-
means model, n_clusters can be any number of clusters
kmeans = KMeans(n_clusters=...)
kmeans.fit(data)
```

Tutorial 8.1: To implement K-means clustering using **sklearn** on a sample data, is as follows:

```
from sklearn.cluster import KMeans
# Load the dataset
data = [[1, 2], [2, 3], [3, 4], [4, 5], [5, 6], [6, 7]]
# Create and fit the k-means model
kmeans = KMeans(n_clusters=3)
kmeans.fit(data)
# Predict the cluster labels for each data point
labels = kmeans.predict(data)
print(f"Clusters labels for data: {labels}")
```

Following is an output which shows the respective cluster label for the above six data:

```
Clusters labels for data: [1 1 2 2 0 0]
```

K-prototype

K-prototype clustering is a generalization of K-means clustering that allows for mixed clusters with both numerical and categorical data. It works by first randomly

selecting K centroids, just like K-means. It then assigns each data point to the nearest centroid. The centroids are then updated to be the mean of the data points in their respective clusters. This process is repeated until the centroids no longer change. It is a used for clustering data that has both numerical and categorical characteristics. And also, for textual data.

For example, K-prototype can be applied here. Imagine you have a social media dataset of users and their posts. You want to group users into clusters based on both their demographic information (e.g., age, gender) and their posting behavior (e.g., topics discussed, sentiment).

Before moving to the tutorials let us look at the syntax for implementing K-prototype with K modes, which is as follows*:*

```
from kmodes.kprototypes import KPrototypes
# Load the dataset
data = ...
# Create and fit the k-prototypes model
kproto = KPrototypes(n_clusters=3, init='Cao')
kproto.fit(data, categorical=[0, 1])
```

Tutorial 8.2: To implement K-prototype using **K modes** on a sample data, is as follows:

```
import numpy as np
from kmodes.kmodes import KModes
# Load the dataset
data = [[1, 2, 'A'], [2, 3, 'B'], [3, 4, 'A'], [4, 5, 'B'], [5, 6, 'B'], [6, 7, 'A']]
# Convert the data to a NumPy array
data = np.array(data)
# Define the number of clusters
num_clusters = 3
```

```
# Create and fit the k-prototypes model
kprototypes = KModes(n_clusters=num_clusters, init='random')
kprototypes.fit(data)
# Predict the cluster labels for each data point
labels = kprototypes.predict(data)
print(f"Clusters labels for data: {labels}")
```

Output:

```
Clusters labels for data: [2 0 2 1 0 2]
```

Hierarchical clustering

Hierarchical clustering is an algorithm that creates a tree-like structure of clusters by merging or splitting groups of data points. There are two main types of hierarchical clustering, that is, agglomerative and divisive. Agglomerative hierarchical clustering starts with each data point in its own cluster and then merges clusters until the desired number of clusters is reached. On the other hand, divisive hierarchical clustering starts with all data points in a single cluster and then splits clusters until the desired number of clusters is reached. It is a versatile algorithm. It can cluster any type of data. Often used in social network analysis to identify communities. Additionally, it is used in data mining to discover hierarchical relationships in data.

For example, hierarchical clustering can be applied here. Imagine you have a network of people connected by friendship ties. You want to group people into clusters based on the strength of their ties.

Before moving to the tutorials let us look at the syntax for implementing hierarchical clustering with sklearn, which is as follows:

```
from sklearn.cluster import AgglomerativeClustering
# Load the dataset
```

```
data = ...
# Create and fit the hierarchical clustering model
hier = AgglomerativeClustering(n_clusters=3)
hier.fit(data)
```

Tutorial 8.3: To implement hierarchical clustering using **sklearn** on a sample data, is as follows:

```
from sklearn.cluster import AgglomerativeClustering
# Load the dataset
data = [[1, 1], [1, 2], [2, 2], [2, 3], [3, 3], [3, 4]]
# Create and fit the hierarchical clustering model
cluster = AgglomerativeClustering(n_clusters=3)
cluster.fit(data)
# Predict the cluster labels for each data point
labels = cluster.labels_
print(f"Clusters labels for data: {labels}")
```

Output:

```
Clusters labels for data: [2 0 2 1 0 2]
```

Gaussian mixture models

Gaussian Mixture Models (GMMs) are a type of soft probabilistic clustering algorithm that models' data as a mixture of Gaussian distributions. Each cluster is represented by a Gaussian distribution, and the algorithm estimates the parameters of these distributions to maximize the likelihood of the data given the model. GMMs are a powerful clustering algorithm that can be used to cluster any type of data that can be modeled by a Gaussian distribution. They are widely used in marketing to segment customers, in finance to detect fraud and in data mining to discover hidden patterns. For example, GMMs can be applied here. Imagine you have a dataset of customer transactions. You want to group customers into clusters

based on their spending patterns.

Before moving to the tutorials let us look at the syntax for implementing Gaussian mixture models with sklearn, which is as follows:

```
from sklearn.mixture import GaussianMixture
# Load the dataset
data = ...
# Create and fit the Gaussian mixture model
gmm = GaussianMixture(n_components=3)
gmm.fit(data)
```

Tutorial 8.4: To implement Gaussian mixture models using **sklearn** on a generated sample data, is as follows:

```
import numpy as np
from sklearn.mixture import GaussianMixture
from sklearn.datasets import make_blobs
# Generate some data
X, y = make_blobs(n_samples=100, n_features=2, centers=3, cluster_std=1.5)
# Create a GMM with 3 components/clusters
gmm = GaussianMixture(n_components=3)
# Fit the GMM to the data
gmm.fit(X)
# Predict the cluster labels for each data point
labels = gmm.predict(X)
print(f"Clusters labels for data: {labels}")
```

Output:

```
Clusters labels for data: [2 0 1 0 0 0 1 2 1 1 0 0 2 2 2 1 0
1 2 1 0 0 2 1 1 1 0 2 1 2 2 1 2 2 2 2 0
 1 1 1 2 0 1 2 0 0 1 1 2 0 1 0 1 0 1 0 1 2 1 0 0 1 1
2 1 2 2 0 0 2 1 0 0 2
```

```
0 1 2 2 0 2 2 2 0 1 2 0 0 0 0 1 2 1 1 0 2 2 1 2 0 2]
```

Principal component analysis

Principal Component Analysis (PCA) is a linear dimensionality reduction algorithm that identifies the principal components of the data. These components represent the directions of maximum variance in the data and can be used to represent the data in a lower-dimensional space. PCA is a widely used algorithm in data visualization, machine learning and signal processing. Principal usage of dimensionality reduction is to decrease the dimensionality of high-dimensional data, such as images or text, and to preprocess data for machine learning algorithms.

For example, PCA can be applied here. Imagine you have a dataset of customer transactions. You want to group customers into clusters based on their spending patterns.

Before moving to the tutorials let us look at the syntax for implementing principal component analysis with **sklearn**, which is as follows:

```
from sklearn.decomposition import PCA
# Load the dataset
data = ...
# Create and fit the PCA model
pca = PCA(n_components=2)
pca.fit(data)
```

Tutorial 8.5: To implement principal component analysis using **sklearn** on an iris flower dataset, is as follows:

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.decomposition import PCA
# Load the Iris dataset
iris = load_iris()
```

```
X = iris.data
# Create a PCA model with 2 components
pca = PCA(n_components=2)
# Fit the PCA model to the data
X_pca = pca.fit_transform(X) #Transform the data into 2 principal components
print(f"Variance explained by principal components: {pca.explained_variance_ratio_}")
```

X_pca is a 2D **numpy** array of shape **(n_samples, 2)**, that contains principal component. Each row represents a sample and each column is a principal component.

Output:

```
Variance explained by principal components: [0.92461872 0.05306648]
```

As output shows first principal component explains 92.46% of the variance in the data. Second principal component explains 5.30% of the variance in the data.

Singular value decomposition

Singular Value Decomposition (SVD) is a linear dimensionality reduction algorithm that decomposes a matrix into three matrices: U, Σ, and V. The U matrix contains the left singular vectors of the original matrix, the Σ matrix contains the singular values of the original matrix, and the V matrix contains the right singular vectors of the original matrix. SVD can be applied to a range of tasks, such as reducing dimensionality, compressing data and extracting features. It is commonly utilized in text mining, image processing and signal processing.

For example, SVD can be applied here. Imagine you have a dataset of customer reviews. You want to summarize the reviews using a smaller set of features.

Before moving to the tutorials let us look at the syntax for

implementing singular value decomposition with sklearn, which is as follows:

```
from numpy.linalg import svd
# Load the dataset
data = ...
# Perform the SVD
u, s, v = svd(data)
```

Tutorial 8.6: To implement singular value decomposition using **sklearn** on an iris flower dataset is as follows:

```
import numpy as np
from sklearn.decomposition import TruncatedSVD
# Load the Iris dataset
iris = load_iris()
X = iris.data
# Create a truncated SVD model with 2 components
svd = TruncatedSVD(n_components=2)
# Fit the truncated SVD model to the data
X_svd = svd.fit_transform(X)
print(f"Variance explained after singular value decompo
sition: {svd.explained_variance_ratio_}")
```

Output:

```
Variance explained after singular value decomposition:
[0.52875361 0.44845576]
```

DBSCAN

Density-Based Spatial Clustering of Applications with Noise (DBSCAN) is a density-based clustering algorithm that identifies groups of data points that are densely packed together. It works by identifying core points, which are points that have a minimum number of neighbors within a specified radius. These core points form the basis of clusters and other points are assigned to clusters based

on their proximity to core points. It is useful when the number of clusters is unknown. Commonly used for data that is not well-separated, particularly in computer vision, natural language processing, and social network analysis.

For example, DBSCAN can be applied here. Imagine you have a dataset of customer locations. You want to group customers into clusters based on their proximity to each other.

Before moving to the tutorials let us look at the syntax for implementing DBSCAN with sklearn, which is as follows:

```
from sklearn.cluster import DBSCAN
# Load the dataset
data = ...
# Create and fit the DBSCAN model
dbscan = DBSCAN(eps=0.5, min_samples=5)
dbscan.fit(data)
```

Tutorial 8.7: To implement DBSCAN using **sklearn** on a generated sample data, is as follows:

```
import numpy as np
from sklearn.cluster import DBSCAN
from sklearn.datasets import make_moons
# Generate some data
X, y = make_moons(n_samples=200, noise=0.1)
# Create a DBSCAN clusterer
dbscan = DBSCAN(eps=0.3, min_samples=10)
# Fit the DBSCAN clusterer to the data
dbscan.fit(X)
# Predict the cluster labels for each data point
labels = dbscan.labels_
print(f"Clusters labels for data: {labels}")
```

Output:

```
Clusters labels for data: [0 0 1 0 1 0 0 1 1 1 0 0 1
1 1 0 1 1 1 0 1 0 0 1 1 0 1 0 0 0 1 1 0 0 1 0 1
 1 0 0 0 0 1 0 1 1 1 0 0 1 1 1 0 1 1 0 0 0 1 1 1 1 0
1 0 1 1 1 0 1 1 0 0 1
 1 1 1 1 1 0 1 1 1 0 0 1 1 1 0 1 1 1 1 0 0 0 0 0 1
1 0 1 0 1 1 0 1 0 1 1 0
 0 0 0 1 0 0 1 1 1 0 0 0 1 1 0 0 1 0 1 1 0 0 1 0 0 1 1
1 0 0 0 1 0 0 0 0 1
 1 0 1 0 0 0 1 0 0 1 1 1 0 0 0 0 0 0 1 0 0 0 0 0 1 1 0 1
 0 1 0 0 1 1 0 0 1
 0 1 0 0 0 1 1 0 0 0 0 1 1 1 1]
```

t-distributed stochastic neighbor embedding

t-Distributed Stochastic Neighbor Embedding (t-SNE) is a nonlinear dimensionality reduction algorithm that maps high-dimensional data points to a lower-dimensional space while preserving the relationships between the data points. It works by modeling the similarity between data points in the high-dimensional space as a probability distribution and then minimizing the Kullback-Leibler divergence between this distribution and a corresponding distribution in the lower-dimensional space. It is often used to visualize high-dimensional data, such as images or text and to pre-process data for machine learning algorithms.

For example, t-SNE can be applied here. Imagine you have a high-dimensional dataset, such as images or text. You want to reduce the dimensionality of the data while preserving as much information as possible.

Before moving to the tutorials let us look at the syntax for implementing t-SNE with sklearn, which is as follows:

```
from sklearn.manifold import TSNE
# Load the dataset
data = ...
```

```
# Create and fit the t-SNE model
tsne = TSNE(n_components=2, perplexity=30)
tsne.fit(data)
```

Tutorial 8.8: To implement t-SNE to reduce four dimensions into two dimensions using **sklearn** on an iris flower dataset, is as follows:

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.manifold import TSNE
# Load the Iris dataset
iris = load_iris()
X = iris.data
# Create a t-SNE model
tsne = TSNE()
# Fit the t-SNE model to the data
X_tsne = tsne.fit_transform(X)
# Plot the t-SNE results
import matplotlib.pyplot as plt
plt.scatter(X_tsne[:, 0], X_tsne[:, 1], c=iris.target)
# Define labels and colors
labels = ['setosa', 'versicolor', 'virginica']
colors = ['blue', 'orange', 'green']
# Create a list of handles for the legend
handles = [plt.plot([],[], color=c, marker='o', ls='')
[0] for c in colors]
# Add the legend to the plot
plt.legend(handles, labels, loc='upper right')
# x and y labels
plt.xlabel('t-SNE dimension 1')
plt.ylabel('t-SNE dimension 2')
```

```
# Title
plt.title('t-SNE visualization of the Iris dataset')
# Show the figure
plt.savefig('TSNE.jpg',dpi=600,bbox_inches='tight')
plt.show()
```

Output:

Plot shows output after t-SNE technique, which reduces the dimensionality of the data from four features (sepal length, sepal width, petal length, and petal width) to two dimensions that can be visualized. Each color corresponds to flower species. It gives an idea of how the data is clustered and how the species are separated in the reduced space.

Following figure shows plot showing cluster of flowers:

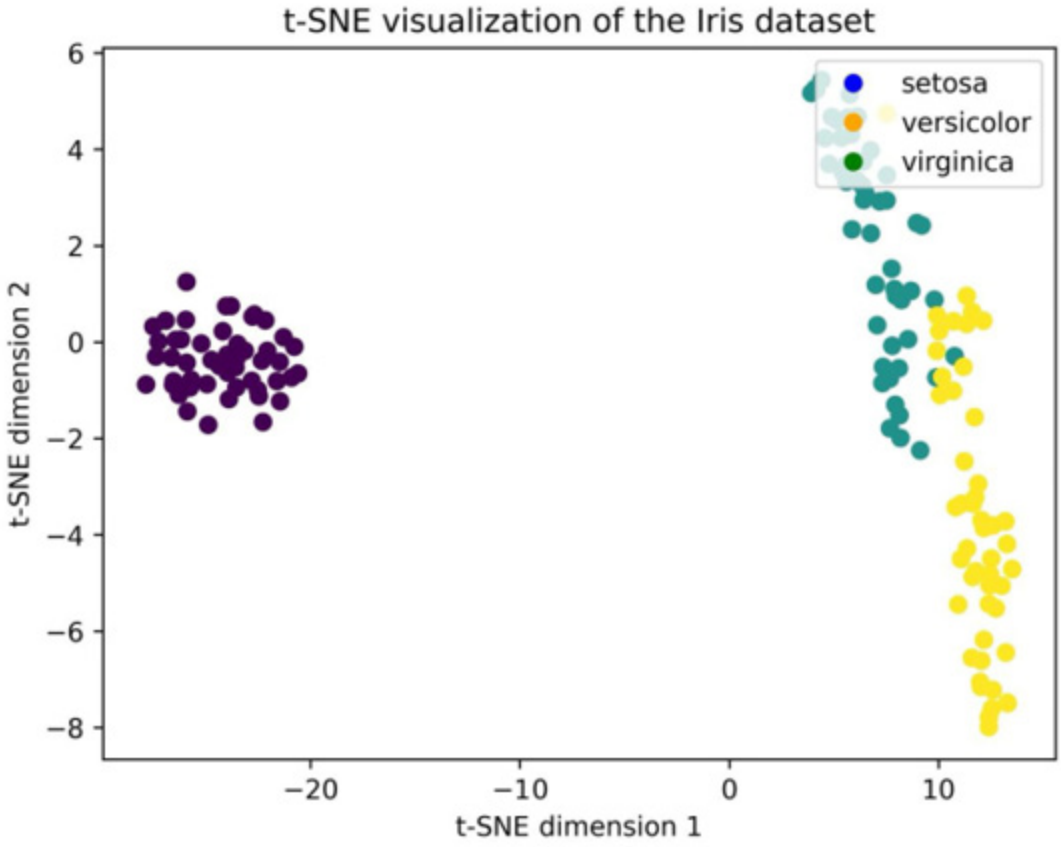

Figure 8.1: *Plot showing cluster of flowers after t-SNE technique on Iris dataset*

Apriori

Apriori is a frequent itemset mining algorithm that identifies frequent item sets in transactional datasets. It works by iteratively finding item sets that meet a minimum support threshold. It is often used in market basket analysis to identify patterns in customer behavior. It can also be used in other domains, such as recommender systems and fraud detection. For example, apriori can be applied here. Imagine you have a dataset of customer transactions. You want to identify common patterns of items that customers tend to buy together.

Before moving to the tutorials let us look at the syntax for implementing Apriori with **apyori** package, which is as follows:

```
from apyori import apriori
# Load the dataset
data = ...
# Create and fit the apriori model
rules = apriori(data, min_support=0.01, min_confidence=0.5)
```

Tutorial 8.9: To implement Apriori to find the all the frequently bought item from a grocery item dataset, is as follows:

```
import pandas as pd
from apyori import apriori
# Load the dataset
data = pd.read_csv(
    '/workspaces/ImplementingStatisticsWithPython/data/chapter7/Groceries.csv')
# Reshape the data from wide to long format
```

```
data = pd.melt(data, id_vars='Channel',
               var_name='Product', value_name='Quantity')
# Group the data by customer and aggregate the product categories into a list
data = data.groupby('Channel')['Product'].apply(list)
# Convert the data into a list of lists
data = data.tolist()
# Create the apriori model
rules = apriori(data, min_support=0.00003)
# Print the rules
for rule in rules:
    for rule in rules:
        print(list(rule.items))
```

Tutorial 8.9 output will display the items in each frequent item set as a list.

Tutorial 8.10: To implement Apriori, to view only the first five frequent items from a grocery item dataset, is as follows:

```
import pandas as pd
from apyori import apriori
# Load the dataset
data = pd.read_csv(
    '/workspaces/ImplementingStatisticsWithPython/data/chapter7/Groceries.csv')
# Reshape the data from wide to long format
data = pd.melt(data, id_vars='Channel',
               var_name='Product', value_name='Quantity')
# Group the data by customer and aggregate the product categories into a list
data = data.groupby('Channel')['Product'].apply(list)
# Convert the data into a list of lists
```

```
data = data.tolist()
# Create the apriori model
rules = apriori(data, min_support=0.00003)
# Print the rules and the first 5 elements
rules = list(rules)
rules = rules[:5]
for rule in rules:
    for item in rule.items:
        print(item)
```

Output:

```
Delicassen
Detergents_Paper
Fresh
Frozen
Grocery
```

Tutorial 8.11: To implement Apriori, to view all most frequent items with the support value of each itemset from the grocery item dataset, is as follows:

```
import pandas as pd
from apyori import apriori
# Load the dataset
data = pd.read_csv(
    '/workspaces/ImplementingStatisticsWithPython/data/chapter7/Groceries.csv')
# Reshape the data from wide to long format
data = pd.melt(data, id_vars='Channel',
               var_name='Product', value_name='Quantity')
# Group the data by customer and aggregate the product categories into a list
data = data.groupby('Channel')['Product'].apply(list)
# Convert the data into a list of lists
```

```
data = data.tolist()
# Create the apriori model
rules = apriori(data, min_support=0.00003)
# Print the rules
for rule in rules:
    # Join the items in the itemset with a comma
    itemset = ", ".join(rule.items)
    # Get the support value of the itemset
    support = rule.support
    # Print the itemset and the support in one line
    print("{}: {}".format(itemset, support))
```

Eclat

Eclat is a frequent itemset mining algorithm similar to Apriori, but more efficient for large datasets. It works by using a vertical data format to represent transactions. It is also used in market basket analysis to identify patterns in customer behavior. It can also be used in other areas such as recommender systems and fraud detection. For example, Eclat can be applied here. Imagine you have a dataset of customer transactions. You want to identify frequent item sets in transactional datasets efficiently.

Tutorial 8.12: To implement frequent item data mining using a sample data set of transactions, is as follows:

```
# Define a function to convert the data from horizontal to vertical format
def horizontal_to_vertical(data):
  # Initialize an empty dictionary to store the vertical format
  vertical = {}
  # Loop through each transaction in the data
  for i, transaction in enumerate(data):
```

```
    # Loop through each item in the transaction
    for item in transaction:
      # If the item is already in the dictionary, append the transaction ID to its value
      if item in vertical:
        vertical[item].append(i)
      # Otherwise, create a new key-value pair with the item and the transaction ID
      else:
        vertical[item] = [i]
  # Return the vertical format
  return vertical
# Define a function to generate frequent item sets using the ECLAT algorithm
def eclat(data, min_support):
  # Convert the data to vertical format
  vertical = horizontal_to_vertical(data)
  # Initialize an empty list to store the frequent item sets
  frequent = []
  # Initialize an empty list to store the candidates
  candidates = []
  # Loop through each item in the vertical format
  for item in vertical:
    # Get the support count of the item by taking the length of its value
    support = len(vertical[item])
    # If the support count is greater than or equal to the minimum support, add the item to the frequent list and the candidates list
    if support >= min_support:
      frequent.append((item, support))
```

```
    candidates.append((item, vertical[item]))
 # Loop until there are no more candidates
 while candidates:
  # Initialize an empty list to store the new candidates
  new_candidates = []
  # Loop through each pair of candidates
  for i in range(len(candidates) - 1):
   for j in range(i + 1, len(candidates)):
    # Get the first item set and its transaction IDs fro
m the first candidate
    itemset1, tidset1 = candidates[i]
    # Get the second item set and its transaction IDs fr
om the second candidate
    itemset2, tidset2 = candidates[j]
    # If the item sets have the same prefix, they can be
 combined
    if itemset1[:-1] == itemset2[:-1]:
     # Combine the item sets by adding the last eleme
nt of the second item set to the first item set
     new_itemset = itemset1 + itemset2[-1]
     # Intersect the transaction IDs to get the support
 count of the new item set
     new_tidset = list(set(tidset1) & set(tidset2))
     new_support = len(new_tidset)
     # If the support count is greater than or equal to t
he minimum support, add the new item set to the freque
nt list and the new candidates list
     if new_support >= min_support:
      frequent.append((new_itemset, new_support))
      new_candidates.append((new_itemset, new_tids
et))
```

```
    # Update the candidates list with the new candidates
    candidates = new_candidates
  # Return the frequent item sets
  return frequent
# Define a sample data set of transactions
data = [
  ["A", "B", "C", "D"],
  ["A", "C", "E"],
  ["A", "B", "C", "E"],
  ["B", "C", "D"],
  ["A", "B", "C", "D", "E"]
]
# Define a minimum support value
min_support = 3
# Call the eclat function with the data and the minimum support
frequent = eclat(data, min_support)
# Print the frequent item sets and their support counts
for itemset, support in frequent:
  print(itemset, support)
```

Output:

```
A 4
B 4
C 5
D 3
E 3
AB 3
AC 4
AE 3
BC 4
```

```
BD 3
CD 3
CE 3
ABC 3
ACE 3
BCD 3
```

FP-Growth

FP-Growth is a frequent itemset mining algorithm based on the FP-tree data structure. It works by recursively partitioning the dataset into smaller subsets and then identifying frequent item sets in each subset. FP-Growth is a popular association rule mining algorithm that is often used in market basket analysis to identify patterns in customer behavior. It is also used in recommendation systems and fraud detection. For example, FP-Growth can be applied here. Imagine you have a dataset of customer transactions. You want to identify frequent item sets in transactional datasets efficiently using a pattern growth approach.

Before moving to the tutorials let us look at the syntax for implementing FP-Growth with **mlxtend.frequent_patterns**, which is as follows:

```
from mlxtend.frequent_patterns import fpgrowth
# Load the dataset
data = ...
# Create and fit the FP-Growth model
patterns = fpgrowth(data, min_support=0.01, use_colnames=True)
```

Tutorial 8.13: To implement frequent item for data mining using FP-Growth using **mlxtend**. frequent patterns, as follows:

```
import pandas as pd
```

```
# Import fpgrowth function from mlxtend library for frequent pattern mining
from mlxtend.frequent_patterns import fpgrowth
# Import TransactionEncoder class from mlxtend library for encoding data
from mlxtend.preprocessing import TransactionEncoder
# Define a list of transactions, each transaction is a list of items
data = [["A", "B", "C", "D"],
  ["A", "C", "E"],
  ["A", "B", "C", "E"],
  ["B", "C", "D"],
  ["A", "B", "C", "D", "E"]]
# Create an instance of TransactionEncoder
te = TransactionEncoder()
# Fit and transform the data to get a boolean matrix
te_ary = te.fit(data).transform(data)
# Convert the matrix to a pandas dataframe with column names as items
df = pd.DataFrame(te_ary, columns=te.columns_)
# Apply fpgrowth algorithm on the dataframe with a minimum support of 0.8
# and return the frequent itemsets with their corresponding support values
fpgrowth(df, min_support=0.8, use_colnames=True)
```

Output:

```
 support itemsets
0 1.0 (C)
1 0.8 (B)
2 0.8 (A)
3 0.8 (B, C)
```

6. 4 0.8 (A, C)

Model selection and evaluation

Unlike supervised learning, unsupervised learning methods commonly use evaluation metrics such as **Silhouette Score (SI), Davies-Bouldin Index (DI), Calinski-Harabasz Index (CI)** and **Adjusted Rand Index (RI)** to check performance and quality of machine learning models.

Evaluation metrices and model selection for unsupervised

Evaluation matrices may vary depending on type of unsupervised learning problem. Although, SI, DI, CI and RI are useful to evaluate the clustering results. The silhouette score measures how well each data point fits into its assigned cluster, based on the average distance to other data points in the same cluster and the nearest cluster. Its score ranges from -1 to 1 with higher values indicating better clustering. DI measures the average similarity between each cluster and its most similar cluster, based on the ratio of intra-cluster distances to inter-cluster distances. The index ranges from zero to infinity with lower values indicating better clustering. CI measures the ratio of the between-cluster variance to the within-cluster variance, based on the sum of the squared distances of the data points to their cluster centroids. The index ranges from zero to infinity, with higher values indicating better clustering. RI measures the similarity between two clusters of the same data set, based on the number of pairs of examples assigned to the same or different clusters in both clustering. Its index ranges from -1 to 1, with higher values indicating better agreement. Here too, the performance, complexity, interpretability and resource requirements remain the selection criteria.

Tutorial 8.14 with snippets, explains how to use some common model selection and evaluation techniques for unsupervised learning.

Tutorial 8.14: To implement a tutorial that illustrates model selection and evaluation in unsupervised machine learning using iris data is as follow:

To begin, we import the required modules and load the iris dataset, by taking all the features only excluding the label, as demonstrated. The aim is to determine the optimal number of clusters from the dataset and assess the result with evaluation matrices as follows:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.cluster import KMeans, AgglomerativeClustering
from sklearn.metrics import silhouette_score, davies_bouldin_score, calinski_harabasz_score, adjusted_rand_score
# Load dataset
iris = load_iris()
X = iris.data # Features
y = iris.target # True labels
print(iris.feature_names)
```

Output:

```
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
```

We define clustering models to compare, including K-means, agglomerative clustering and define SI, DI, CI and RI metrics to evaluate the models, as follows:

```
# Define candidate models
```

```
models = {
    'K-means': KMeans(),
    'Agglomerative Clustering': AgglomerativeClustering()
}
# Evaluate models using multiple metrics
metrics = {
    'Silhouette score': silhouette_score,
    'Davies-Bouldin index': davies_bouldin_score,
    'Calinski-Harabasz index': calinski_harabasz_score,
    'Adjusted Rand index': adjusted_rand_score
}
```

To evaluate the quality of each cluster, we fit the model and plot the results.

```
# Fit model, get cluster labels, compare reasults
scores = {}
for name, model in models.items():
    labels = model.fit_predict(X)
    scores[name] = {}
    for metric_name, metric in metrics.items():
        if metric_name == 'Adjusted Rand index':
            score = metric(y, labels) # Compare true labels and predicted labels
        else:
            score = metric(X, labels) # Compare features and predicted labels
        scores[name][metric_name] = score
        print(f'{name}, {metric_name}: {score:.3f}')
# Plot scores
fig, ax = plt.subplots(2, 2, figsize=(10, 10))
for i, metric_name in enumerate(metrics.keys()):
```

```
    row = i // 2
    col = i % 2
    ax[row, col].bar(scores.keys(), [score[metric_name] for score in scores.values()])
    ax[row, col].set_ylabel(metric_name)
# Save the figure
plt.savefig('Clustering_model_selection_and_evaluation.png', dpi=600, bbox_inches='tight')
```

Output:

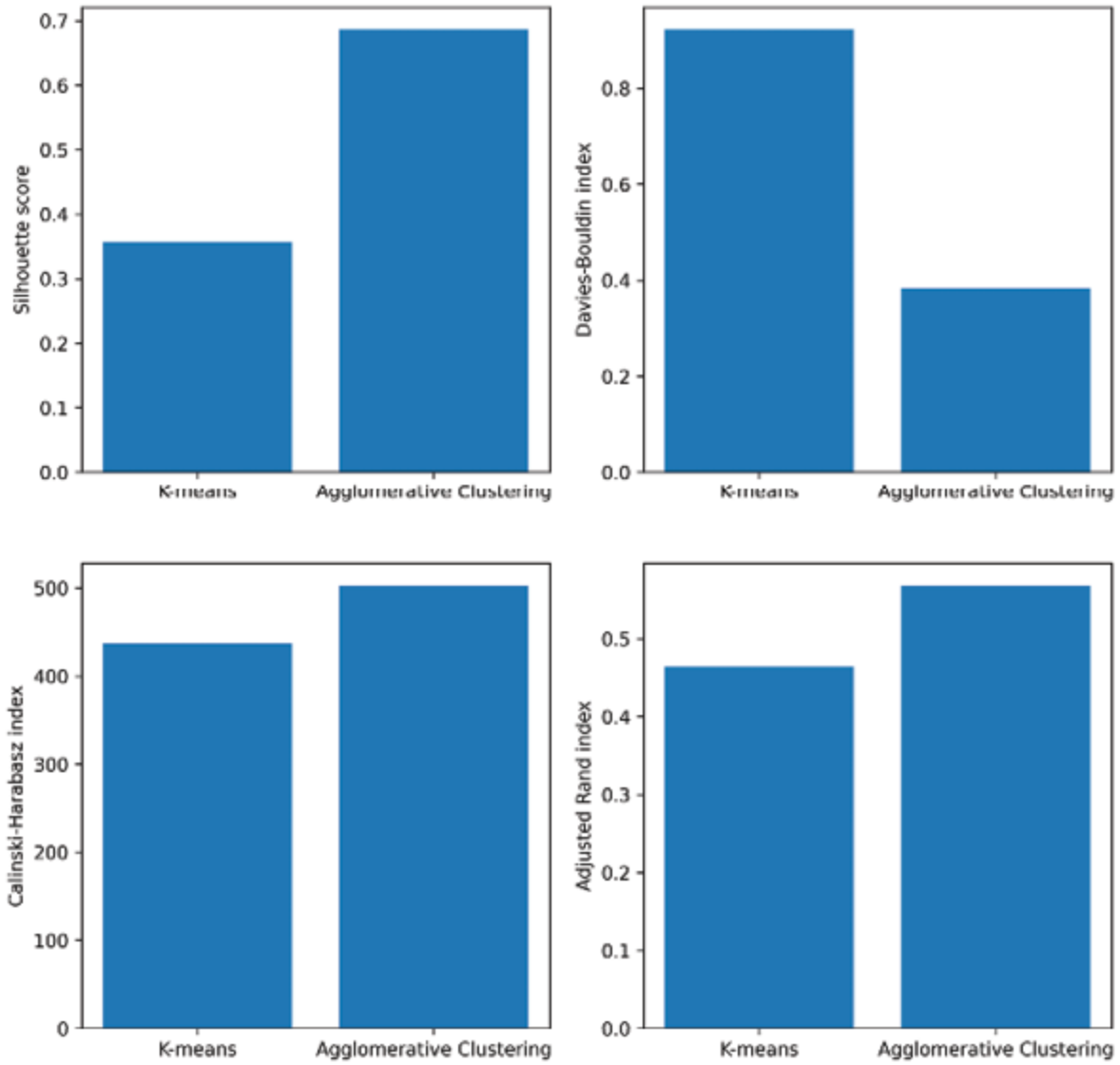

Figure 8.2. *Plot comparing the SI, CI, DI, RI scores of different unsupervised algorithms on the iris dataset*

Figure 8.2 and the SI, CI, DI, RI scores show that agglomerative clustering performs better than K-means on the iris dataset according to all four metrics. Agglomerative clustering has a higher SI score, which means that the clusters are more cohesive and well separated. It also has a lower DI, which means that the clusters are more distinct and less overlapping. In addition, agglomerative clustering has a higher CI score, which means that the clusters have a higher ratio of inter-cluster variance to intra-cluster variance. Finally, agglomerative clustering has a higher RI, which means that the predicted labels are more consistent with the true labels. Therefore, agglomerative clustering is a better model choice for this data.

Conclusion

In this chapter, we explored unsupervised learning and algorithms for uncovering hidden patterns and structures within unlabeled data. We delved into prominent clustering algorithms like K-means, K-prototype, and hierarchical clustering, along with probabilistic approaches like Gaussian mixture models. Additionally, we covered dimensionality reduction techniques like PCA and SVD for simplifying complex datasets. This knowledge lays a foundation for further exploration of unsupervised learning's vast potential in various domains. From customer segmentation and anomaly detection to image compression and recommendation systems, unsupervised learning plays a vital role in unlocking valuable insights from unlabeled data.

We hope that this chapter has helped you understand and apply the concepts and methods of statistical machine learning, and that you are motivated and inspired to learn more and apply these techniques to your own data and

problems.

The next *Chapter 9, Linear Algebra, Nonparametric Statistics, and Time Series Analysis* explores time series data, linear algebra and nonparametric statistics.

Join our book’s Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com